Bab 8

Memori Virtual

POKOK BAHASAN:

- ✓ Latar Belakang
- ✓ Demand Paging
- ✓ Page Replacement
- ✓ Alokasi Frame
- ✓ Thrashing
- ✓ Contoh Sistem Operasi

TUJUAN BELAJAR:

Setelah mempelajari materi dalam bab ini, mahasiswa diharapkan mampu:

- ✓ Memahami latar belakang memori virtual
- ✓ Memahami maksud demand paging
- ✓ Memahami mekanisme page replacement
- ✓ Memahami algoritma alokasi frame
- ✓ Mengetahui implementasi memori virtual

8.1 LATAR BELAKANG

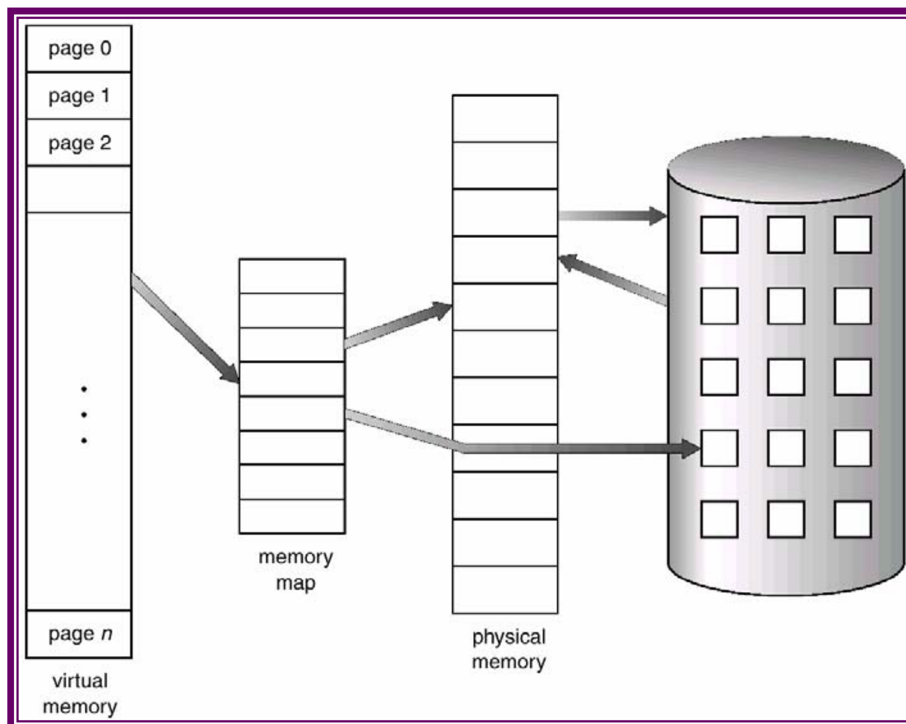
Sebagian besar algoritma manajemen memori memerlukan satu kebutuhan dasar yaitu instruksi yang akan dieksekusi harus berada di memori fisik. Pada beberapa kasus, keseluruhan program tidak diperlukan. Misalnya :

- Program mempunyai kode untuk menangani kondisi error yang tidak biasa. Karena error-error ini jarang terjadi, kode ini hampir tidak pernah dieksekusi.
- Array, list dan tabel dialokasikan lebih dari kapasitas memori yang diperlukan
- Pilihan dan gambaran program jarang digunakan

Pada kasus dimana keseluruhan program dibutuhkan, mungkin tidak semua diperlukan pada saat yang sama. Kemampuan mengeksekusi program hanya pada beberapa bagian dari memori mempunyai beberapa keuntungan yaitu :

- Program tidak terbatas jumlah memori fisik yang tersedia sehingga user dapat menulis program untuk ruang alamat virtual yang sangat besar yang berarti menyederhanakan programming task.
- Karena setiap program user dapat menggunakan memori fisik yang lebih kecil, pada waktu yang sama dapat menjalankan lebih banyak program.
- I/O yang lebih sedikit diperlukan untuk load atau swap program user ke memori, sehingga setiap program user dapat berjalan lebih cepat.

Memori virtual adalah teknik yang memisahkan memori logika user dari memori fisik. Menyediakan memori virtual yang sangat besar diperuntukkan untuk programmer bila tersedia memori fisik yang lebih kecil. Programmer tidak perlu khawatir jumlah memori fisik yang tersedia, sehingga dapat berkonsentrasi pada permasalahan pemrograman. Gambaran memori virtual dapat dilihat pada Gambar 8-1.

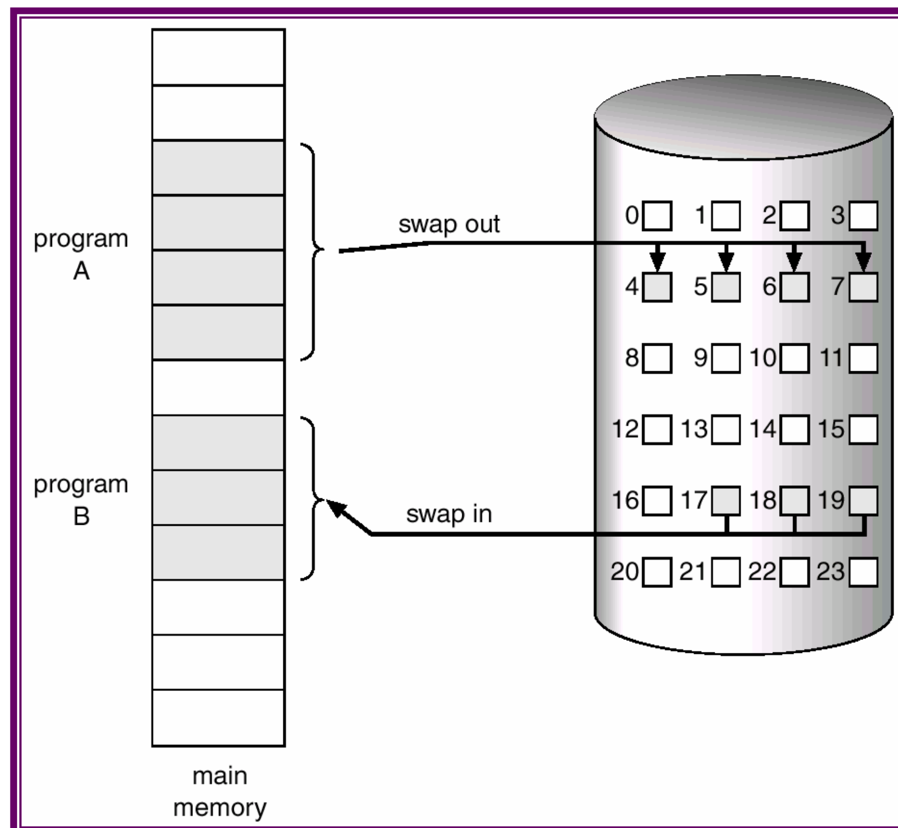


Gambar 8-1 : Memori virtual lebih besar daripada memori fisik

Memori virtual biasanya diimplementasikan menggunakan *demand paging* atau *demand segmentation* juga digunakan. Tetapi algoritma *segment-replacement* lebih kompleks daripada algoritma *page-replacement* karena segmen mempunyai ukuran yang bervariasi.

8.2 DEMAND PAGING

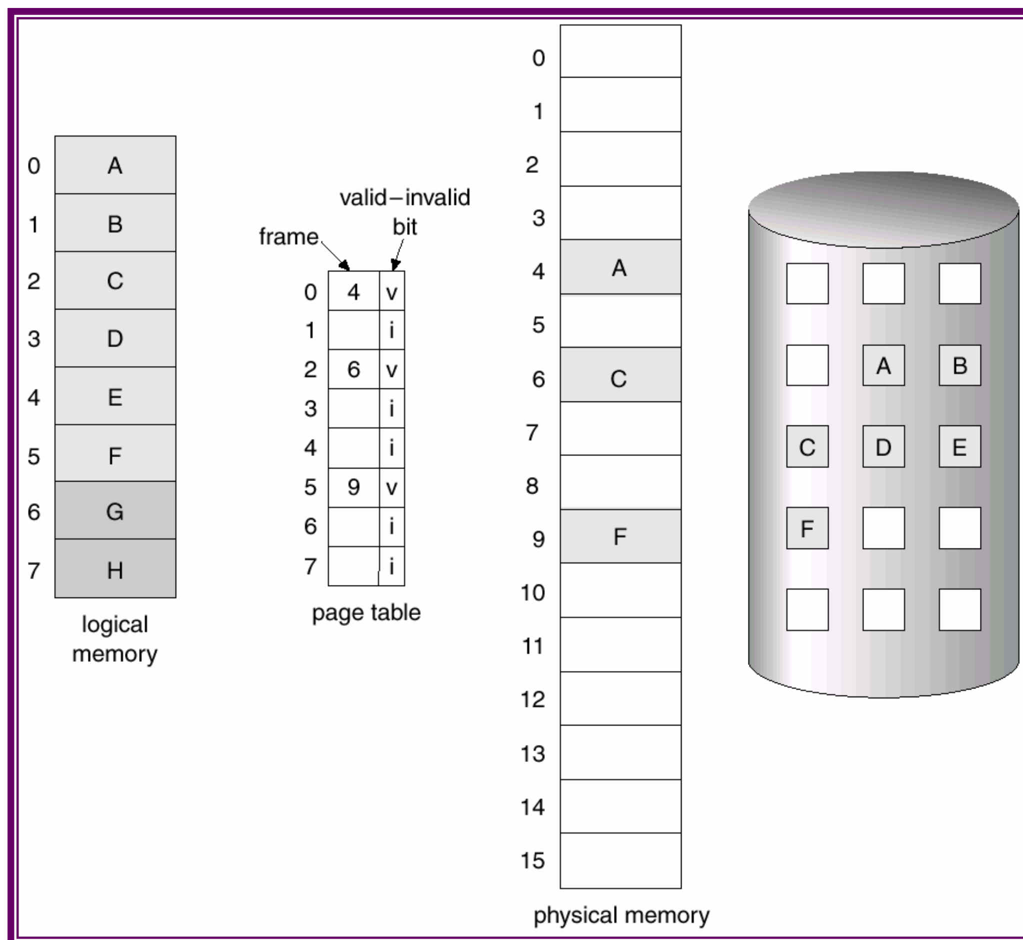
Demand paging adalah sistem *paging* dengan *swapping* seperti pada Gambar 8-2. Page diletakkan di memori hanya jika diperlukan. Hal ini menyebabkan kebutuhan I/O lebih rendah, kebutuhan memori lebih rendah, respon lebih cepat dan lebih banyak user yang menggunakan.



Gambar 8-2 : Sistem paging dengan swapping

Proses disimpan di memori sekunder (disk). Jika proses akan dieksekusi, maka dipindah (swap) ke memori. Menggunakan *lazy swapper* untuk melakukan *swapping* bila *page* tersebut akan digunakan yang berarti sebuah *page* tidak pernah ditukar ke memori kecuali *page* diperlukan. Jika *page* diperlukan, dilakukan acuan ke *page* tersebut, tetapi jika acuan *invalid* maka dilakukan penghentian. *Page* yang sedang tidak berada di memori tersebut akan dibawa ke memori dari *backing store*.

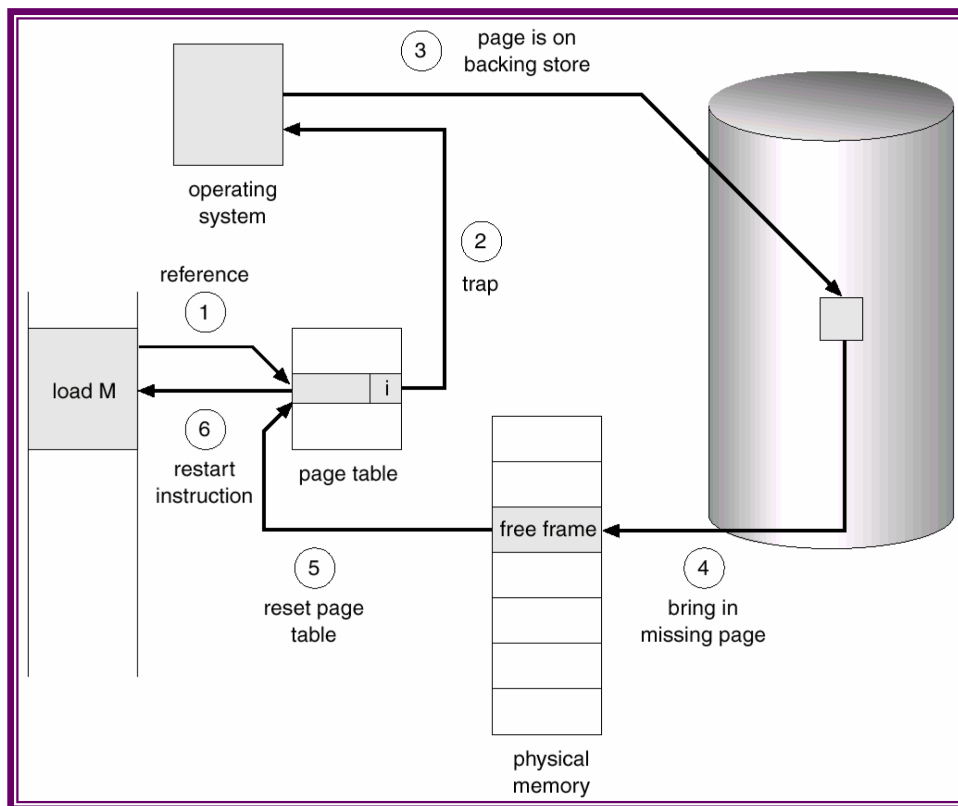
Untuk membedakan antara *page* pada memori dengan *page* pada disk digunakan valid-invalid bit. Tabel *page* untuk *page* yang berada di memori diset "valid', sedangkan tabel *page* untuk *page* yang tidak sedang di memori (ada pada disk) diset "invalid" seperti Gambar 8-3.



Gambar 8-3 : Beberapa page tidak sedang berada di memori

Akses ke *page* yang diset "invalid" menyebabkan *page fault*, yang menyebabkan trap ke sistem operasi. Karena status (register, kode kondisi, counter instruksi) dari proses ter-interrupt disimpan bila terjadi *page fault*, proses dapat dimulai lagi pada tempat dan status yang sama, kecuali *page* yang cocok sedang di memori dan sedang diakses. Prosedur untuk menangani *page fault* seperti Gambar 8-4 sebagai berikut :

1. Sistem operasi melihat tabel untuk menentukan jika acuan invalid maka proses dihentikan dan page sedang tidak berada di memori.
2. Jika acuan invalid dilakukan trap ke sistem operasi.
3. Sistem mencari *frame* kosong
4. Sistem melakukan proses swapping ke *frame* bebas.
5. Tabel *page* di-reset, bit valid-invalid diset 1 atau valid
6. instruksi di-restart.



Gambar 8-4 : Langkah-langkah bila terjadi page fault

Apabila tidak ditemukan *frame* bebas maka dilakukan *page replacement* yaitu mencari beberapa *page* di memori yang tidak digunakan kemudian dilakukan *swap out* ke backing store. Terdapat beberapa algoritma *page replacement* dimana performansi algoritma diharapkan menghasilkan jumlah *page fault* minimum. Beberapa *page* kemungkinan dibawa ke memori beberapa kali.

Perangkat keras yang dibutuhkan untuk mendukung *demand paging* sama dengan perangkat keras untuk sistem *paging* dengan *swapping* yaitu

- Tabel page : tabel mempunyai kemampuan untuk memberi entry bit valid-invalid atau nilai khusus untuk bit proteksi
- Memori sekunder : digunakan untuk membawa *page* yang tidak di memori dan biasanya adalah disk kecepatan tinggi yang disebut *swap device*.

8.3 PERFORMANSI DEMAND PAGING

Demand paging memberikan efek yang signifikan dalam kinerja sistem computer. Diasumsikan *ma* adalah access time ke memori dan *p* adalah probabilitas terjadi *page fault* ($0 \le p \le 1$), maka *effective access time* didefinisikan sebagai :

$$EAT = (1-p) \times ma + p \times \text{page_fault-time}$$

Untuk menghitung *effective access time*, harus diketahui berapa waktu yang diperlukan untuk melayani *page fault*. *Page fault* menyebabkan terjadi

1. Trap ke sistem operasi.
2. Menyimpan register dan status proses.
3. Menentukan interrupt adalah *page fau.t*
4. Memeriksa *page* acuan legal atau tidak dan menentukan lokasi *page* pada disk.
5. Membaca dari disk ke *frame* bebas :
   a. Menunggu di antrian untuk perangkat sampai permintaan membaca dilayani.
   b. Menunggu perangkat mencari dan / atau waktu latency.
   c. Memulai transfer dari *page* ke *frame* bebas.
6. Sementara menunggu, alokasikan CPU untuk user lain.
7. Interrupt dari disk (melengkapi I/O).
8. Menyimpan register dan status process user lain.

9. Menentukan interrupt dari disk.
10. Memperbaiki tabel *page* dan tabel lain untuk menunjukkan *page* yang dimaksud sudah di memori.
11. Menunggu CPU dialokasikan untuk proses ini kembali.
12. Menyimpan kembali register, status proses dan tabel *page* baru, kemudian melanjutkan kembali instruksi yang di-interupsi.

Tidak semua langkah diatas diperlukan pada setiap kasus. Pada beberapa kasus, terdapat tiga komponen utama dari waktu pelayanan *page fault* yaitu

1. Melayani interrupt *page fault.*
2. Membaca *page*.
3. Memulai kembali proses.

Untuk menghitung *effective access time* dari sistem demand paging perhatikan contoh berikut. Diasumsikan memory access 100 ns. Rata-rata waktu latency untuk hard disk adalah 8 ms, waktu pencarian 15 ms dan rata-rata transfer sebesar 1 ms. Total waktu paging ≈ 25 ms.

$$\begin{aligned} \textit{Effective access time} &= (1-p) \text{ x } (100) + p \text{ x } (25 \text{ ms}) \\ &= (1-p) \text{ x } 100 + p \text{ x } 25000000 \\ &= 100 + 24999900 \text{ x } p \end{aligned}$$

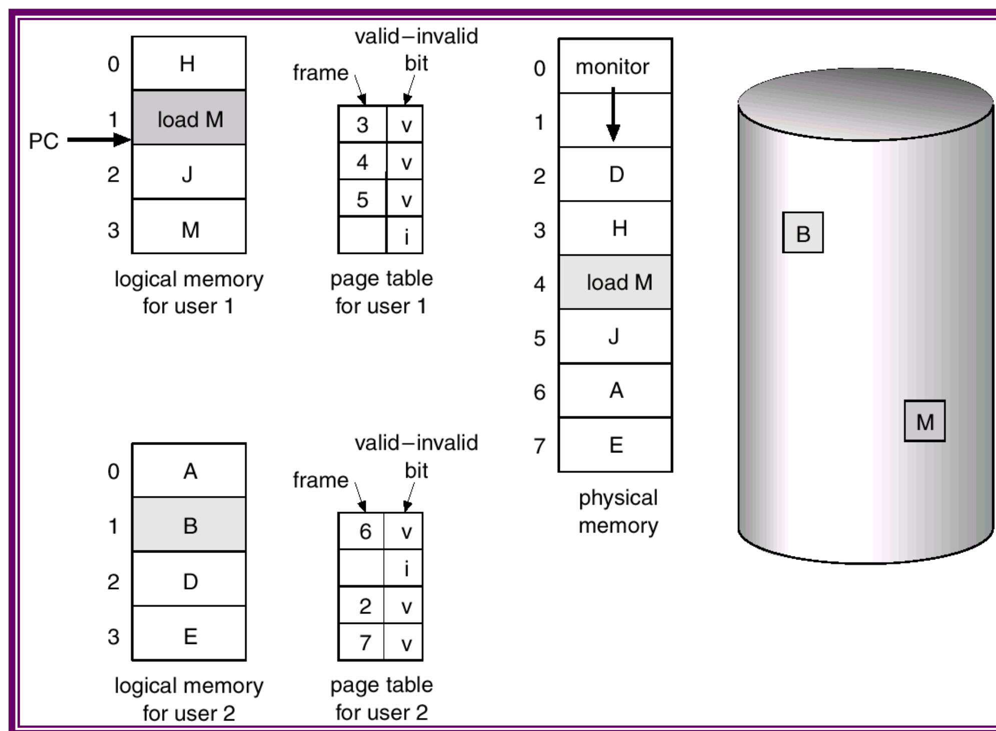
Apabila satu dari 1000 akses menyebabkan page fault, maka *effective access time* = 25 micro-sec (lebih lambat dengan faktor 250). Tetapi bila menginginkan degradasi kurang dari 10% maka

$$\begin{aligned} &110 > 100 + 25000000 \text{ x } p \\ &10 > 250000000 \text{ x } p \\ &p < 0.0000004 \end{aligned}$$

Perlu diperhatikan system harus mempertahankan rata-rata page-fault yang rendah pada sistem *demand-paging*. Sebaliknya, jika *effective access time* meningkat maka akan memperlambat eksekusi proses secara drastis.

8.4 PAGE REPLACEMENT

Page replacement diperlukan pada situasi dimana proses dieksekusi perlu *frame* bebas tetapi tidak tersedia frame *bebas*. Sistem harus menemukan satu *frame* yang sedang tidak digunakan dan membebaskannya. Untuk membebaskan *frame* dengan cara menulis isinya untuk ruang swap dan mengubah tabel page (dan tabel lain) yang menunjukkan page tidak lagi di memori. Kebutuhan akan *page replacement* dapat dilihat pada Gambar 8-5.



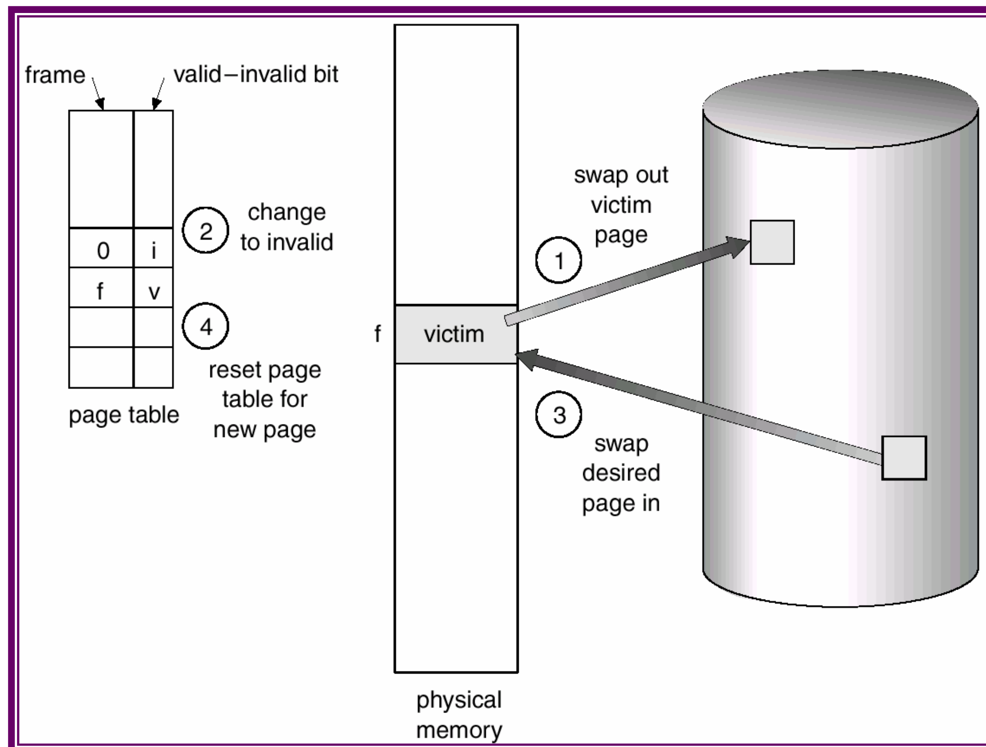
Gambar 8-5 : Kebutuhan akan page replacement

Langkah-langkah untuk *page fault* yang memerlukan *page replacement* seperti Gambar 8-6 adalah sebagai berikut :

1. Carilah lokasi *page* yang diharapkan pada disk.
2. Carilah *frame* kosong dg cara :
   - Bila ada *frame* kosong, gunakan.

- Bila tidak ada, gunakan algoritma *page replacement* untuk menyeleksi *frame* yang akan menjadi korban.
- Simpan *page* korban ke disk, ubah tabel *page*.

3. Baca *page* yang diinginkan ke *frame* kosong yang baru, ubah tabel *page*.
4. Mulai kembali proses user.



Gambar 8-6 : Langkah-langkah page replacement

8.5 ALGORITMA PAGE REPLACEMENT

Terdapat beberapa algoritma *page replacement*, setiap sistem operasi mempunyai skema yang unik. Algoritma *page replacement* secara umum diinginkan yang mempunyai rata-rata *page fault* terendah. Algoritma dievaluasi dengan menjalankannya pada string tertentu dari memory reference dan menghitung jumlah *page fault*. String yang mengacu ke memori disebut *reference string* (string acuan). String acuan dibangkitkan secara random atau dengan menelusuri sistem dan

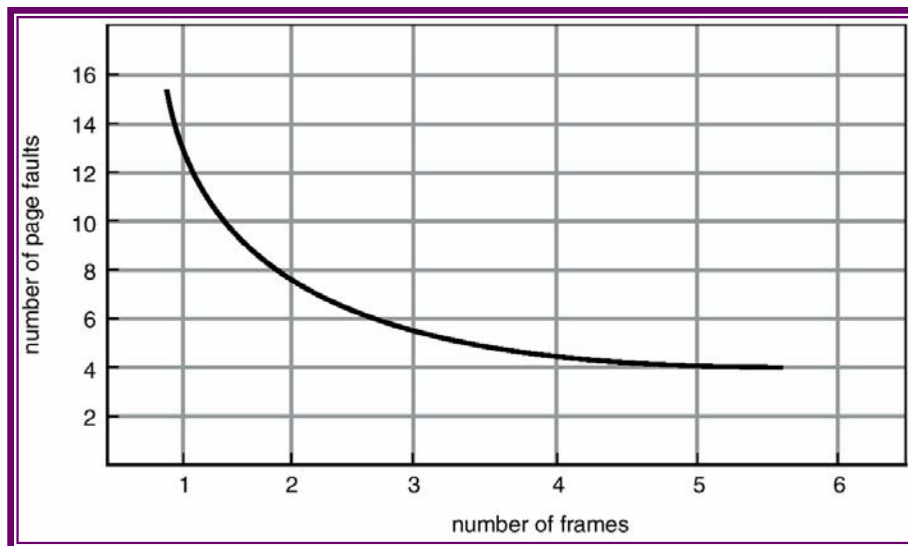
menyimpan alamat dari memori acuan. Misalnya jika ditelusuri proses tertentu, disimpan alamat berikut :

0100, 0432, 0101, 0612, 0102, 0103, 0104,
0101, 0611, 0102, 0103, 0104, 0101, 0610,
0102, 0103, 0104, 0101, 0609, 0102, 0105

dimana 100 byte per *page* direduksi ke string acuan :

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Untuk menentukan jumlah *page fault* untuk string acuan dan algoritma *page-replacement* tertentu, harus diketahui jumlah *page frame* tersedia juga harus diketahui. Semakin tinggi jumlah frame lebih tinggi, semakin rendah jumlah page fault. Hal ini dapat dilihat dengan grafik pada Gambar 8-7.



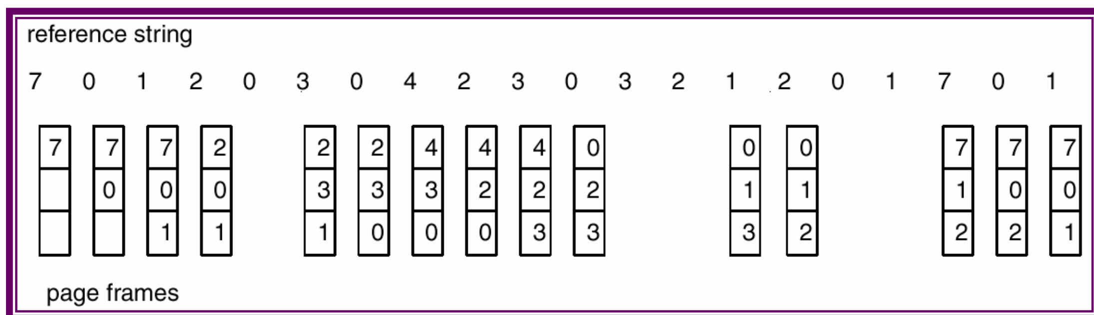
Gambar 8-7 : Grafik jumlah page fault terhadap jumlah frame

Terdapat beberapa algoritma *page replacement* antara lain algoritma *first in first our* (FIFO), optimal dan *least recently use* (LRU). Pada sub bab berikut akan diilustrasikan algoritma *page replacement* tersebut dengan menggunakan string acuan

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

8.5.1. Algoritma FIFO

Algoritma FIFO merupakan algoritma paling sederhana. Algoritma FIFO diasosiasikan dengan sebuah *page* bila *page* tersebut dibawa ke memori. Bila ada suatu *page* yang akan ditempatkan, maka posisi *page* yang paling lama yang akan digantikan. Algoritma ini tidak perlu menyimpan waktu pada saat sebuah *page* dibawa ke memori. Ilustrasi algoritma FIFO dapat dilihat pada Gambar 8-8.



Gambar 8-8 : Algoritma page replacement FIFO

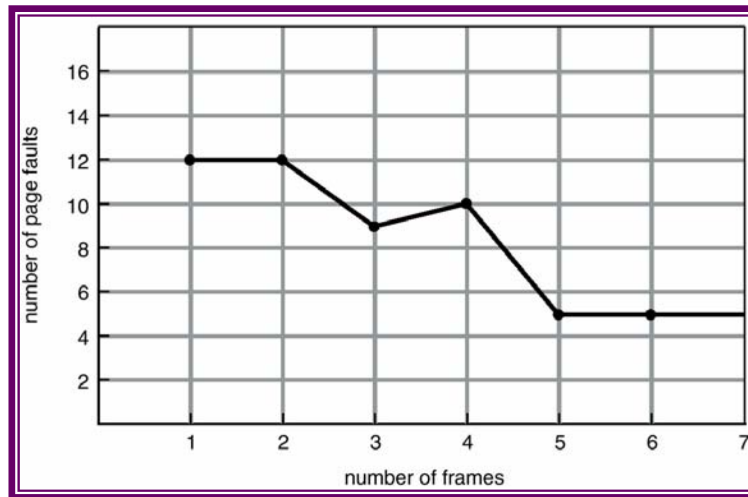
Meskipun algoritma FIFO mudah dipahami dan diimplementasikan, performansi tidak selalu bagus karena algoritma FIFO menyebabkan *Belady's anomaly*. *Belady's anomaly* mematahkan fakta bahwa untuk beberapa algoritma *page replacement*, bila rata-rata *page fault* meningkat, akan meningkatkan jumlah alokasi *frame*. Sebagai contoh, jika menggunakan string acuan :

1, 2, 3, 4, 1, 2, 5, 1, 2, 5, 1, 2, 3, 4, 5

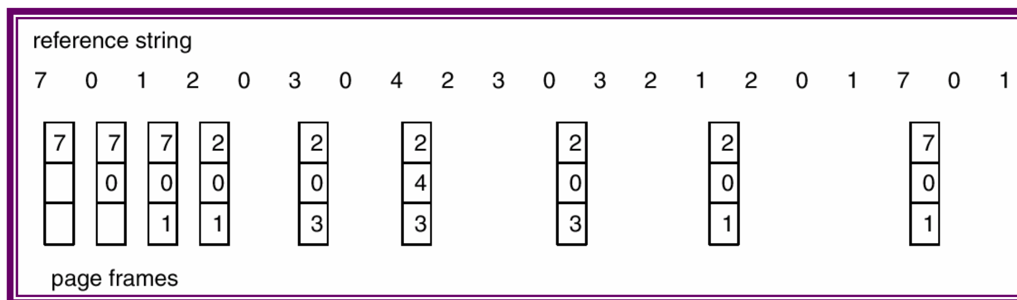
dengan algoritma FIFO terjadi *Belady's anomaly* seperti Gambar 8-9.

8.5.2. Algoritma Optimal

Algoritma optimal merupakan hasil penemuan dari *Belady's anomaly*. Algoritma ini mempunyai rata-rata *page fault* terendah. Algoritma optimal akan mengganti *page* **yang tidak akan** digunakan untuk periode waktu terlama. Algoritma ini menjamin rata-rata *page fault* terendah untuk jumlah *frame* tetap tetapi sulit implementasinya. Ilustrasi dari algoritma optimal dapat dilihat pada Gambar 8-10.



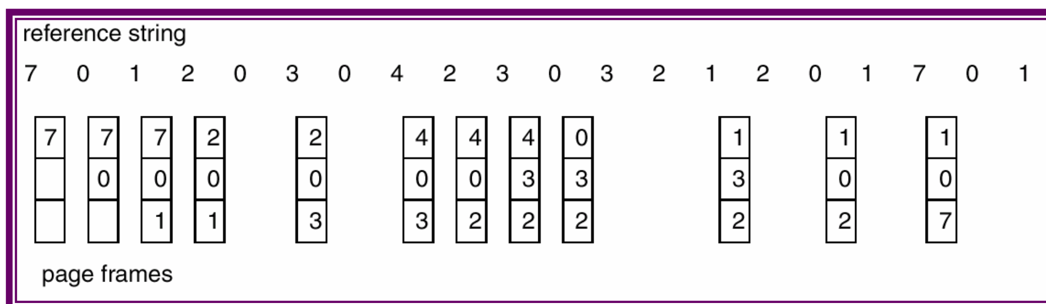
Gambar 8-9 : Belady's anomaly



Gambar 8-10 : Algoritma page replacement optimal

8.5.3. Algoritma Least Recently Use (LRU)

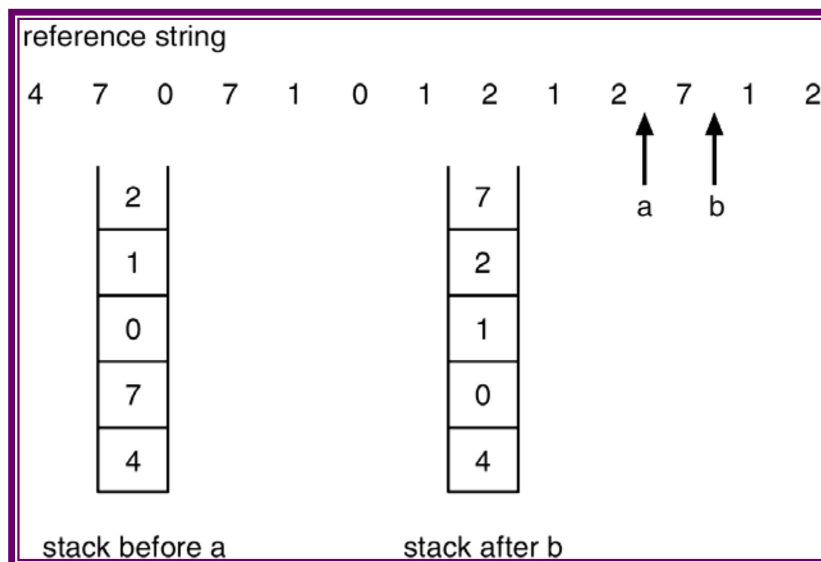
Algoritma LRU merupakan perpaduan dari algoritma FIFO dan optimal. Prinsip dari algoritma LRU adalah mengganti *page* **yang sudah tidak digunakan** untuk periode waktu terlama. Ilustrasi algoritma LRU dapat dilihat pada Gambar 8-11.



Gambar 8-11 : Algoritma page replacement LRU

Untuk mengimplementasikan algoritma LRU, digunakan dua model yaitu :

- **Counter** : setiap entry tabel *page*e diasosiasikan dengan sebuah "time-of-use" dan sebuah clock logika atau counter ditambahkan ke CPU. Clock ini dinaikkan untuk setiap acuan ke memori. Jika sebuah acuan ke *page* dibuat, isi clock register di-copy ke *time-of-use* pada tabel *page* untuk *page* tersebut.
- **Stack** : stack dari nomor *page* diatur. Jika sebuah *page* digunakan acuan, maka *page* dihapus dari stack dan meletakkan pada *top of stack*. Dengan cara ini, stack selalu digunakan *page* dan bagian bawah untuk *page* LRU. Implementasi stack untuk algoritma LRU diilustrasikan pada Gambar 8-12.



Gambar 8-12 : Implementasi LRU menggunakan stack

8.6 ALOKASI FRAME

Alokasi frame berhubungan dengan mekanisme alokasi sejumlah memori bebas yang tetap diantara beberapa proses. Meskipun terdapat beberapa variasi pengalokasian *frame* bebas ke beberapa proses, tetapi strategi dasar jelas yaitu : proses user dialokasikan untuk sembarang *frame* bebas.

Jumlah minimum *frame* per proses ditentukan oleh arsitektur dimana jumlah maksimum tergantung jumlah memori fisik yang tersedia. Jumlah minimim *frame*

ditentukan oleh arsitektur *instruction-set*. Bila terjadi *page fault* sebelum eksekusi instruksi selesai, instruksi harus di-restart. Sehingga tersedia *frame* yang cukup untuk membawa semua *page* yang berbeda dimana sembarang instruksi dapat mengacu. Misalnya mikrokomputer menggunakan memori 128K yang dikomposisikan dengan *page* ukuran 1K, maka terbentuk 128 *frame*. Jika sistem operasi menggunakan 35K, maka 93 *frame* sisa digunakan program user. Bila suatu program menyebabkan *page fault* sebanyak 93 kali, maka menempati 93 *frame* bebas tersebut. Jika terjadi *page fault* ke 94, dari 93 *frame* yang terisi harus dipilih salah satu untuk diganti yang baru. Bila program selesai, 93 *frame* tersebut dibebaskan kembali.

Terdapat 2 bentuk algoritma alokasi yaitu *equal allocation* dan *proportional allocation*. Pada *equal allocation*, jika terdapat *m frame* dan *n* proses, maka setiap proses dialokasikan sejumlah *frame* yang sama (*m/n frame*). Pada *proportional allocation* setiap proses dialokasikan secara proporsional berdasarkan ukurannya. Jika ukuran virtual memori untuk proses p_i adalah s_i dan total jumlah frame yang tersedia m, maka *frame* ke a_i dapat dialokasikan ke proses p_i sama dengan :

$$a_i = s_i / S \times m$$

Dimana S = $\sum s_i$. Contohnya :

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Selain itu terdapat algoritma alokasi berprioritas yang menggunakan skema proporsional dengan lebih melihat prioritas proses daripada ukuran proses. Jika proses Pi membangkitkan *page fault,* dipilih satu dari *frame-frame* dari proses yang mempunyai nomor prioritas terendah.

8.7 ALOKASI GLOBAL DAN ALOKASI LOKAL

Page replacement adalah faktor terpenting lain yang harus dipertimbangkan dalam alokasi *frame*. Pada multiple process yang berkompentisi mendapatkan *frame*, algoritma *page replacement* dikelompokkan dalam 2 kategori yaitu *global replacement* dan *local replacement.*

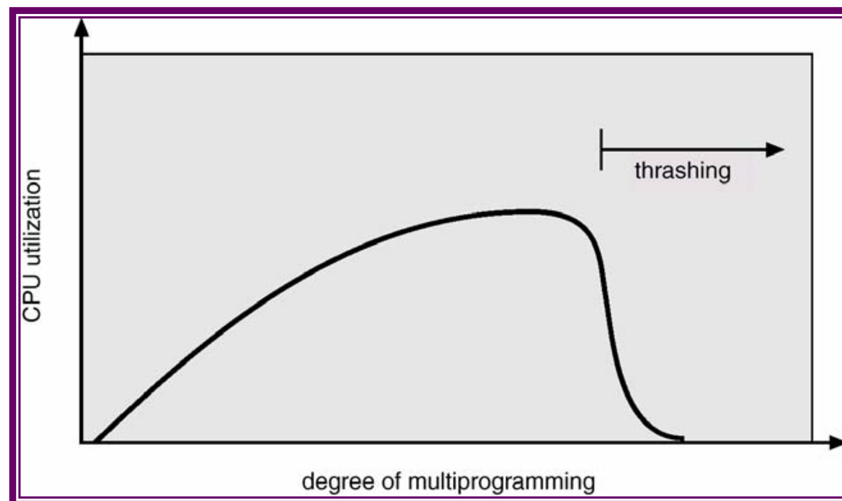
Global replacement mengijinkan suatu proses untuk menyeleksi suatu *frame* yang akan dipindah dari sejumlah *frame*, meskipun *frame* tersebut sedang dialokasikan ke proses yang lain. Pada **local replacement,** jumlah frame yang dialokasikan untuk proses tidak berubah. Setiap proses dapat memilih dari *frame-frame* yang dialokasikan untuknya.

Permasalahan pada *global replacement* adalah proses tidak dapat mengontrol rata-rata page fault. Sejumlah page pada memori untuk sebuah proses tidak hanya tergantung pada perilaku paging untuk proses tersebut, tetapi juga perilaku paging untuk proses yang lain. Bagaimanapun, karena algoritma global replacement menghasilkan throughput yang lebih besar, metode ini sering digunakan.

8.8 THRASHING

Misalnya sembarang proses tidak mempunyai *frame* yang cukup. Meskipun secara teknis dapat mengurangi jumlah *frame* yang dialokasikan sampai minimum, terdapat sejumlah *page* yang sedang aktif digunakan. Jika suatu proses tidak memiliki jumlah *frame* yang cukup, maka sering terjadi *page fault*. Sehingga harus mengganti beberapa *page*. Tetapi karena semua *page* sedang digunakan, harus mengganti *page* yang tidak digunakan lagi kemudian. Konsekuensinya, sering terjadi *page fault* lagi dan lagi. Proses berlanjut *page fault*, mengganti *page* untuk *page fault* dan seterusnya.

Kegiatan aktifitas paging yang tinggi disebut *thrashing.* Sebuah proses mengalami *thrashing* jika menghabiskan lebih banyak waktu untuk paging daripada eksekusi. Efek *thrashing* dapat dibatasi dengan menggunakan algoritma local (priority) replacement. Grafik terjadinya proses *thrashing* pada sistem multiprogramming dapat dilihat pada Gambar 8-13.



Gambar 8-13 : Thrashing

RINGKASAN:

LATIHAN SOAL :

1. Diketahui sistem memory demand paging. Page table menggunakan register. Membutuhkan 8 milisecond untuk melayani page fault jika frame kosong tersedia atau page yang di-replace tidak dimodifikasi dan 20 milisecond jika page yang di-replace dimodifikasi. Waktu akses memori adalah 100 nanosecond. Diasumsikan page yang di-replace akan dimodifikasi adalah 70 persen dari waktu. Berapa rata-rata page fault yang diterima untuk *effective access time* tidak lebih dari 200 nanosecond ?
2. Diketahui string acuan dari page :

   1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

   Berapa banyak page fault yang terjadi untuk algoritma page replacement berikut dengan satu, dua, tiga, empat, lima, enam atau tujuh frame ? Ingat bahwa semua frame diinisialisasi kosong, sehingga setiap page unik pertama akan bernilai masing-masing satu fault

   a. LRU

   b. FIFO

c. Optimal

3. Diketahui array 2 dimensi *A* sebagai berikut :

```
var A : array[1..100] of array[1..100] of integer
```

Dimana *A*[1][1] berada pada lokasi 200 pada sistem page memory dengan page-page berukuran 200. Suatu proses kecil pada page 0 (lokasi 0 s/d 199) untuk manipulasi matriks, sehingga setiap instruksi dimulai dari page 0.

Untuk 3 frame page, berapa banyak page fault yang dibangkitkan oleh loop inisialisasi array berikut menggunakan LRU dan asumsi frame page 1 sudah terdapat proses dan 2 frame page lainnya diinisialisasi kosong.

a.
```
For (j = 1; j <= 100; j++)
        For (i = 1; i <= 100; i++)
                A[i][j] = 0;
```

b.
```
For (i = 1; i <= 100; i++)
        For (j = 1; j <= 100; j++)
                A[i][j] := 0;
```

4. Diketahui sistem demand paging dengan paging disk mempunyai waktu akses dan transfer rata-rata 20 milisec. Alamat ditranslasikan melalui page table di memory, dengan waktu akses 1 microsec per akses memory. Sehingga acuan ke memori melalui page table sama dengan 2 kali akses memory. Untuk memperbaiki waktu, ditambahkan associative memory yang menurunkan waktu akses menjadi satu acuan memori, jika entri page table berada di associative memory. Diasumsikan 80 % akses pada associative memory dan dari sisanya (20%), 10% nya (atau 2 persen dari total) menyebabkan page fault. Berapakah effective access time-nya?